第9章 内联函数

本章概要

- 在C中，保持效率的一个方法是使用宏，这通过预处理器来实现。然而宏有其内在的问题。
- 本章将介绍预处理宏所存在的问题，并介绍如何用内联函数解决这些问题

预处理器的缺陷

- 宏看上去像一个函数调用，但他并非真正的函数调用，这样就隐藏了难以发现的错误。
- 例一：对于下面的宏定义：
  - #define F (x) (x+1)

  假如有下面的一个调用：
  - F(1)

  展开会是什么样的代码？　　(x) (x+1)(1)

预处理器的缺陷

- 例二：
  - #define FLOOR(x,b) x>b?0:1

  对于下面的函数调用，展开后会有什么问题：

  If(FLOOR(a & 0x0f, 0x7)){……}

  →If(a & 0x0f >= 0x07 ? 0: 1)

  是否正确？

  可以将宏定义改为如下形式：

  #define FLOOR(x,b) ((x)>=(b)?0:1)

预处理器的缺陷

- 例三：

```
//: C09:MacroSideEffects.cpp
#include "../require.h"
#include <fstream>
using namespace std;

#define BAND(x) (((x)>5 && (x)<10) ? (x) : 0)

int main() {
  ofstream out("macro.out");
  assure(out, "macro.out");
  for(int i = 4; i < 11; i++) {
    int a = i;
    out << "a = " << a << endl << '\t';
    out << "BAND(++a)=" << BAND(++a) << endl;
    out << "\t a = " << a << endl;
  }
} ///:~
```

内联函数(inline function)

- 内联函数是真正的函数，能够像普通函数一样具有我们期望的行为。
- 唯一与普通函数的不同之处在于，内联函数在适当的地方可以像宏一样展开，减少了函数调用的开销。
- 实质是用存储空间来换取时间
- 在C++中，应该尽可能避免使用宏，只使用内联函数。

内联函数

- 下面是用inline关键字定义了一个内联函数：
  - inline int nextNum(int x) { return ++x; }

  编译器将与对待普通函数一样，检查参数类型和返回值类型，做必要的转换。
- 如果需要在多处使用内联函数，内联函数必须定义在头文件中。

类成员作为内联函数

- 在类的内部，如果希望将类的成员定义为内联函数，inline关键字不是必要的，只需要将成员的实现直接定义在类的定义中即可。
- C09:Inline.cpp
- 是否应该将所有的成员函数都以内联的形式定义？

成员内联函数：访问函数

- 在类中，内联函数最重要的用途之一是用作访问函数(access function)。这是一些小函数，它允许读写对象的状态。
- C09:Access.cpp
- 访问函数可以大大提高代码的效率。
- 访问器和修改器：
  - C09:Rectangle2.cpp
  - C09:Cpptime.h
  - C09:Cpptime.cpp

带内联函数的Stash和Stack

- C09： Stash4.h/Stash4.cpp
- C09： Stack4.h/Stack4.cpp

内联函数和编译器

- 编译器如何处理内联函数
  - 编译器在遇到内联函数时，像处理正常的函数一样，进行语法和类型的检查，并将函数和函数体放入符号表
  - 当有代码调用内联函数时，编译器首先保证调用正确，即所有的参数和返回值的类型必须满足：要么与函数参数表中的类型一样，要么编译器能够将其转换为正确类型，并且返回值在目标表达式里应该是正确类型或可改变为正确类型。
  - 如果所有的类型信息符合调用的上下文的话，内联函数的代码就会直接替换函数的调用，这样就消除了调用的开销。

内联函数和编译器：限制

- 假如函数太复杂，编译器将放弃内联。因为这时内联可能无法提高效率。
- 假如要显式或隐式地获取函数的地址，编译器也将放弃内联。
- 对于虚函数，如果真正使用了多态(late binding)，编译器也无法执行内联。

内联函数和编译器：限制

- 内联仅仅是给编译器的一个建议，并不是强迫编译器内联。编译器根据实际情况确定是否执行内联。一个好的编译器可以智能地确定内联小的函数，和放弃对复杂函数的内联。
- 由于构造函数和析构函数的复杂性，通常这两个函数不建议定义为内联

减少混乱

- 将类和接口放在一起，会减少类的可读性。
- 建议使用如下的方法定义内联函数：
  - C09:Noinstitu.cpp

预处理器的更多特征

- 内联函数并不能完全代替预处理宏指令。
  - 字符串化(stringizing)
    - 使用字符串化操作符#，可以将标识符转化为字符串。
    - #define DEBUG(x) cout << #x " = " << x << endl
  - 标志粘贴(token pasting)
    - 标志粘贴操作符##允许把两个标识符粘贴在一起自动产生一个新的标识符。

```
#define FIELD(a) char* a##_string; int a##_size
class Record {
  FIELD(one);
  FIELD(two);
  FIELD(three);
  // ...
};
```

改进的错误检查

- 使用内联的断言：
  - require.h
  - C09:ErrTest.cpp

小结

- 由于预处理指令存在的问题，C++中引入了内联函数
- 内联函数的定义方式
  - inline
  - 通常定义在头文件中
- 编译器如何处理内联函数
- 内联函数无法替代的预处理功能